Модули и пакеты

Модули и пакеты

Модули Python — единицы организации программ наивысшего уровня, которые упаковывают программный код и данные для многократного использования и предоставляют изолированные пространства имен, сводящие к минимуму конфликты имен переменных внутри программ. Модули обычно соответствуют файлам программ Python. Каждый файл является модулем, а модули импортируют другие модули, чтобы задействовать определяемые в них имена. Модули могут также соответствовать расширениям, написанным на внешнем языке, таком как C, Java или C#, и даже каталогам в импортированных пакетах. Модули обрабатываются с помощью двух операторов и одной важной функции.

`import`	Позволяет клиенту (импортеру) извлечь модуль как единое целое.
`from`	Позволяет клиентам извлекать отдельные имена из модуля.
`imp.reload` (`reload` в Python 2.X)	Предоставляет способ перезагрузки кода модуля, не останавливая Python.

Для чего используются модули?

- ## Многократное использование кода

  код в файлах модулей постоянен — его можно перезагружать и повторно запускать столько раз, сколько нужно. Не менее важно и то, что модули представляют собой место для определения имен, известных как атрибуты, на которые могут ссылаться многочисленные внешние клиенты. При правильном применении в результате поддерживается модульная конструкция программ, группирующая функциональность в многократно используемые единицы.

- ## Разбиение пространства имен системы

  Модули также считаются организационной единицей наивысшего уровня в программах Python. Во многом подобно локальным областям видимости функций такое решение помогает избежать конфликтов имен в программах. На самом деле вам даже не удастся обойти данную особенность — абсолютно все “существует” в модуле, и запускаемый код, и создаваемые объекты всегда неявно заключаются в модули. По указанной причине модули представляют собой естественные инструменты для группирования компонентов системы.

Для чего используются модули?

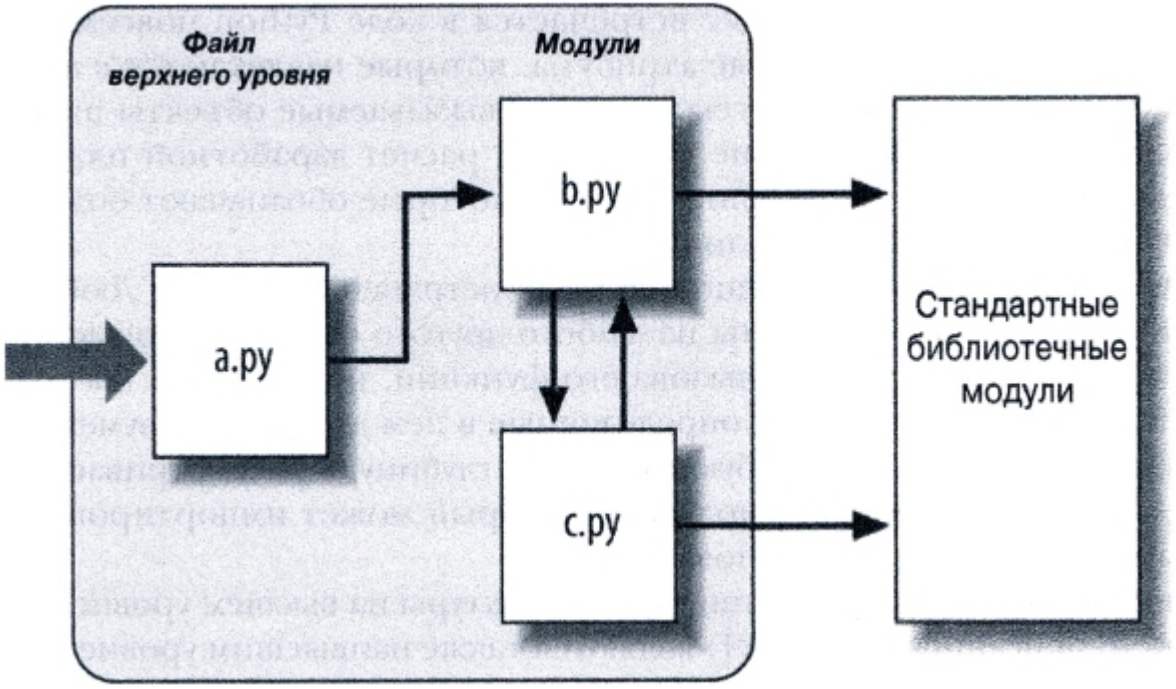
- Реализация разделяемых служб или данных

  С эксплуатационной точки зрения модули также удобны для реализации компо нентов, которые разделяются в рамках системы и потому требуют только одной копии. Например, если необходимо предоставить глобальный объект, применяемый в нескольких функциях или файлах, тогда его можно реализовать в модуле, который затем будет импортироваться многими клиентами.

Архитектура программы Python

- На самом базовом уровне программа Python состоит из текстовых файлов, содержащих операторы Python, с одним главным файлом верхнего уровня и нулем или большим количеством добавочных файлов, известных как модули.
- Файл верхнего уровня (он же сценарий) содержит главный поток управления программы — именно данный файл запускается для старта приложения. Файлы модулей являются библиотеками инструментов, используемых для сбора компонентов, которые применяются файлом верхнего уровня и возможно где-нибудь еще. Файлы верхнего уровня используют инструменты, определенные в файлах модулей, а модули применяют инструменты, определенные в других модулях.
- Несмотря на то что файлы модулей тоже относятся к файлам кода, они обычно ни чего не делают, когда запускаются напрямую; взамен в них определяются инструмен ты, предназначенные для использования в других файлах. Файл импортирует модуль для получения доступа к определенным в нем инструментам, которые известны как его атрибуты — имена переменных, присоединяемые к таким объектам, как функции. В конечном счете, для работы с инструментами мы импортируем модули и обращаемся к их атрибутам.

Архитектура программы Python



Архитектура программы Python

```
def spam(text):                 # Файл b.py
    print(text, 'spam')

import b                        # Файл a.py
b.spam('gumby')                 # Выводит gumby spam
```

Как работает импортирование

1. Поиск файл модуля.
2. Компиляция его в байт-код (при необходимости)
3. Выполнение кода модуля для создания объектов, которые в нем определены.

Путь поиска модулей

1. Домашний каталог программы.
2. Каталоги PYTHONPATH (если установлены).
3. Каталоги стандартной библиотеки.
4. Содержимое любых файлов .pth (при их наличии).
5. Подкаталог site-packages, где размещаются сторонние расширения.

Путь поиска модулей

Домашний каталог (устанавливается автоматически)

- Сначала Python ищет импортируемый файл в домашнем каталоге. Смысл данного элемента зависит от того, каким образом вы запускаете код. Когда вы запускаете программу, данный элемент представляет собой каталог, который содержит файл сценария верхнего уровня программы. Когда вы работаете в интерактивной подсказке, данным элементом является каталог, где производится работа (т. е. текущий рабочий каталог).
- Из-за того, что поиск в домашнем каталоге всегда происходит первым, если программа расположена целиком в единственном каталоге, тогда все ее операции импортирования будут работать автоматически, не требуя какого-либо кон фигурирования путей. С другой стороны, поскольку поиск в этом каталоге про изводится первым, его результаты также переопределяются модулями с теми же самыми именами в других местах пути.

Каталоги PYTHONPATH (допускают конфигурирование)

- Затем Python выполняет поиск во всех каталогах, перечисленных в переменной среды PYTHONPATH, слева направо (при условии, что вы вообще установили ее: она не устанавливается автоматически). Выражаясь кратко, PYTHONPATH — просто список определяемых пользователем и специфичных к платформе имен каталогов, которые содержат файлы кода Python. Вы можете добавить все каталоги, из которых должна быть возможность импортирования, и Python расширит путь поиска модулей, включив в него все перечисленные в переменной PYTHONPATH каталоги.

Каталоги стандартной библиотеки (устанавливаются автоматически)

- Далее Python автоматически выполняет поиск в каталогах, в которых установлены стандартные библиотечные модули. Поскольку поиск в них происходит всегда, они обычно не требуют добавления в переменную PYTHONPATH или включения в файлы конфигурации путей

Каталоги в файлах конфигурации путей .pth (допускают конфигурирование)

- Затем менее употребляемая возможность Python позволяет пользователям добавлять каталоги в путь поиска модулей, просто перечисляя их по одному в строке внутри текстового файла с расширением .pth (от path — путь). Такие файлы конфигурации путей являются несколько более развитым средством, связанным с установкой; мы не будем здесь раскрывать их полностью, но отметим, что они предлагают альтернативу настройкам переменной PYTHONPATH.
- Текстовые файлы с именами каталогов, помещенные в подходящий каталог, могут служить примерно той же роли, что и переменная среды PYTHONPATH. Скажем, если вы работаете в среде Windows с установленной версией Python 3.7, тогда для расширения пути поиска модулей можете поместить файл по имени myconfig.pth на верхний уровень каталога с установленной копией Python (C:\Python37) или в подкаталог site-packages стандартной библиотеки (C:\Python33\Lib\site-packages). В Unix-подобных системах этот файл может находиться в /usr/local/lib/python3.7/site-packages или /usr/local/lib/site-python.

Каталог Lib\ site-packages сторонних расширений

- Наконец, Python автоматически добавляет в путь поиска модулей подкаталог site-packages своей стандартной библиотеки. По соглашению он представляет собой место, в которое устанавливается большинство сторонних расши рений, часто автоматически посредством служебных инструментов distutils,

Источники модулей

- В настоящее время оператор import вида import b способен загрузить или распознать:
- файл исходного кода по имени b. py;
- файл байт-кода по имени b.pyc;
- файл оптимизированного байт-кода по имени b.pyo (менее распространенный Формат);
- каталог по имени b для операций импортирования пакетов
- скомпилированный модуль расширения, написанный на C, C++ или другом языке и динамически связываемый при импортировании (например, b.so в Linux либо b. dll или b. pyd в Cygwin и Windows);
- скомпилированный встроенный модуль, написанный на C и статически связанный с Python;
- компонент в файле ZIP, который автоматически извлекается при импортировании;
- образ в памяти для фиксированных исполняемых файлов;
- класс Java в версии Jython языка Python;
- компонент .NET в версии IronPython языка Python.

Создание модулей

- Чтобы определить модуль, наберите в своем текстовом редакторе какой-нибудь код Python и сохраните его в текстовом файле с расширением . ру; любой файл такого рода автоматически считается модулем Python. Все имена, присвоенные на верхнем уровне модуля, становятся его саприбутами (именами, ассоциированными с объектом модуля) и экспортируются для применения клиентами — переменные автоматически превращаются в атрибуты объекта модуля.

```
def printer(x):    # Атрибут модуля
    print(x)
```

Использование модулей

- Оператор import

```
>>> import module1    # Получить модуль как единое целое (один или больше)
>>> module1.printer('Hello world!')    # Уточнить, чтобы получить имена
Hello world!
```

имя module1 служит двум разным целям — оно идентифицирует внешний файл, подлежащий загрузке, и оно становится переменной в сценарии, которая ссылается на объект модуля после того, как файл загружен

Оператор from

```
>>> from module1 import printer    # Копировать переменную (одну или более)
>>> printer('Hello world!')        # Уточнение не требуется
Hello world!
```

- Такая форма from позволяет указывать одно или несколько имен для копирования, разделенных запятыми. Здесь оператор from имеет такой же эффект, как в предыдущем примере, но из-за того, что импортированное имя копируется в область видимости, где находится from, использование этого имени в сценарии сопряжено с меньшим объемом набора — мы можем работать с именем напрямую, не задавая включающий модуль. В действительности мы обязаны поступать так; from не создает переменную с именем самого модуля.

Оператор from *

```
>>> from module1 import *    # Копировать все переменные
>>> printer('Hello world!')
Hello world!
```

Формально операторы import и from вызывают ту же самую операцию импортирования; форма from * просто добавляет дополнительный шаг, который копирует все имена из модуля в импортирующую область видимости. Она по существу сворачивает пространство имен одного модуля внутрь другого; опять-таки совокупный эффект для нас — меньший объем набора.

Операции импортирования происходят только однократно

simple.py:

```
print('hello')
spam = 1            # Инициалировать переменную
```

```
% python
>>> import simple    # Первая операция импортирования: загружает
                     #   и выполняет код файла
hello
>>> simple.spam     # Присваивание создает атрибут
1
```

```
>>> simple.spam = 2    # Изменение атрибута в модуле
>>> import simple      # Просто извлекает объект уже загруженного модуля
>>> simple.spam        # Код модуля повторно не выполняется:
                       #   атрибут остался неизменившимся
2
```

import и from являются присваиваниями

как и *def*, операторы *import* и *from* являются явными присваиваниями'.
• оператор import присваивает одиночному имени объект целого модуля;
• оператор from присваивает одному или нескольким именам объекты с такими же именами из другого модуля.

small .py:

```
x = 1
y = [1, 2]
% python
>>> from small import x, y # Копировать два имени
>>> x = 42                 # Изменяет только локальное имя x
>>> y[0] = 42              # Модифицирует изменяемый объект на месте


>>> import small          # Получить имя модуля (from этого не позволяет)
>>> small.x               # x из small - не то же, что x здесь
1
>>> small.y               # Но мы разделяем модифицированный изменяемый объект
[42, 2]
```

Межфайловое изменение имен

```
% python
>>> from small import x, y      # Скопировать два имени
>>> x = 42                      # Изменяет только локальное имя x

>>> import small                # Получить имя модуля
>>> small.x = 42                # Изменяет x в другом модуле
```

Эквивалентность import и from

- Оператор from только копирует имена из одного модуля в другой; он не присваивает само имя модуля. По крайней мере, концептуально оператор from следующего вида:

```
from module import name1, name2     # Копировать указанные два имени (только)

 import module                        # Извлечь объект модуля
 name1 = module.name1                 # Копировать имена присваиванием
 name2 = module.name2
 del module                           # Избавиться от имени модуля
```

Когда оператор import обязателен

```
# M.py
def func():
    ...делать что-то...
```

```
# M.py
def func():
    ...делать что-то...
```

```
# O.py
from M import func
from N import func    # Переписывает имя func, извлеченное из M
func()                # Вызывает только N.func!
```

```
# O.py
import M, N     # Получить модули целиком, не только их имена
M.func()        # Теперь мы можем обращаться к обоим именам
N.func()        # Указание имен модулей обеспечивает уникальность
```

```
# O.py
from M import func as mfunc     # Переименование с помощью as
from N import func as nfunc
mfunc(); nfunc()                # Вызов одного или другого
```

Файлы генерируют пространства имен

- Операторы модуля выполняются при его первом импортировании. Когда модуль впервые импортируется в любом месте системы, Python создает пустойобъект модуля и выполняет операторы из файла модуля друг за другом от началадо конца файла.
- Присваивания на верхнем уровне модуля создают его атрибуты. Во время вы полнения операции импортирования не вложенные в def или class операторы на верхнем уровне файла, которые присваивают значения именам (например, =, def), создают атрибуты объекта модуля; присвоенные имена сохраняются в пространстве имен модуля.
- Доступ к пространству имен модуля можно получить через атрибут __dict__ или посредством dir (M). Пространства имен модулей, созданные операциями импортирования, представляют собой словари; к ним можно обращаться через встроенный атрибут__dict__ , ассоциированный с объектами модулей, и инспектировать с помощью функции dir. Функция dir является приблизительным эквивалентом отсортированного списка ключей атрибута__dict__ объекта мо дуля, но содержит унаследованные имена для классов, может возвращать неполный список, а также предрасположена к изменениям от выпуска к выпуску.

Файлы генерируют пространства имен

- Модуль представляет собой одиночную область видимости (локальная является глобальной). Как выяснилось, имена на верхнем уровне модуля следуют таким же правилам ссылки/присваивания, как имена в функции, но локальная и глобальная области видимости совпадают. Выражаясь более формально, они следуют правилу поиска в областях видимости LEGB

Файлы генерируют пространства имен

module2.py:

```
print('starting to load...')    # начало загрузки...
import sys
name = 42

def func(): pass

class klass: pass

print('done loading.')          # загрузка окончена.
```

```
>>> import module2
starting to load...
done loading.
>>> module2.sys
<module 'sys' (built-in)>
>>> module2.name
42
>>> module2.func
<function func at 0x000000000222E7B8>
>>> module2.klass
<class 'module2.klass'>
```

Уточнение имен атрибутов

Простые переменные
X означает поиск имени X в текущих областях видимости (следуя правилу LEGB).

Уточнение
X. Y означает поиск X в текущих областях видимости, затем поиск атрибута Y в объекте X (не в областях видимости).

Пути уточнения
X.Y.Z означает поиск имени Y в объекте X, затем поиск Z в объекте X. Y.

Всеобщность
Уточнение работает для всех объектов с атрибутами: модулей, классов, типов расширений C и т.д.

Импортирование или области видимости

moda .py

```
X = 88              # Переменная X: глобальная по отношению только к данному файлу
def f():
    global X        # Изменить X из этого файла
    X = 99          # Нельзя видеть имена из других модулей
```

modb.py

```
X = 11              # Переменная X: глобальная по отношению только к данному файлу

import moda         # Получить доступ к именам в moda
moda.f()            # Устанавливает moda.X, но не X из этого файла
print(X, moda.X)
```

```
% python modb.py
11 99
```

- функции не могут видеть имена в других функциях, если только они физически не вкладываются в них;
- код модуля не может видеть имена в других модулях, если только он явно не импортирует их

Вложение пространств имен

mod3. py

```
X = 3
```

mod2 .py

```
X = 2
import mod3

print(X, end=' ')     # Собственное глобальное имя X
print(mod3.X)         # X из mod3
```

mod1 .py

```
X = 1
import mod2

print(X, end=' ')          # Собственное глобальное имя X
print(mod2.X, end=' ')     # X из mod2
print(mod2.mod3.X)         # X из вложенного mod3
```

Вложение пространств имен

```
% python mod1.py
2 3
1 2 3
```

mod2 внутри mod1 — это всего лишь имя, которое ссылается на объект с атрибутами, часть которых может ссылаться на другие объекты с атрибутами (import является присваиванием). Для путей вроде mod2 .mod3 .X интерпретатор Python просто производит оценку слева направо, попутно извлекая атрибуты из объектов. Обратите внимание, что в modi можно записать import mod2 и затем mod2.mod3.X, но нельзя записать import mod2 .mod3 — такой синтаксис инициирует то, что называется импортированием пакетов (каталогов). Импортирование пакетов также создает вложение пространств имен модулей, но его операторы import служат для отражения деревьев каталогов, а не простых цепочек импортирования файлов.